ISSN:0975-9646

Integration Testing in AOSE Methodologies

Manikkanan. D[#], Prasanna Kumar.K[#], Vagheesan. K[*], Venkataramana. D[*]

[#]*Department of IT, Adhiparasakthi Engineering college, Melmaruvathur, Tamilnadu, India*
[*]*Department of CSE, Pondicherry Engineering College, Puducherry, India*

Abstract— **As agents gain acceptance as a technology there is a growing need for practical methods for developing agent applications. Agent Oriented Software Engineering (AOSE) methodologies were proposed to develop complex distributed system based upon the agent paradigm. AOSE has several methodologies that focus only on Software Development Life Cycle (SDLC) phases such as analysis phase and design phase. Very few methodologies include implementation phase also. Although agent technology is gaining world wide popularity, a hindrance to its uptake is the lack of proper testing mechanisms for agent based systems. The main objective of our paper is to incorporate testing phase in existing AOSE methodology named Prometheus. Recent survey claims that unit testing framework is been developed for Prometheus methodologies and our work extends by developing integrated testing framework.**

Keywords- ***Agent-Oriented Software Engineering, Multi-Agent Systems, Unit Testing, Integration Testing.***

I. Introduction

A software development methodology refers to the framework that is used to structure, plan, and control the process of developing a software system [1]. A wide variety of such frameworks have evolved over the years, each with its own recognized strength and weaknesses. Now an increasing number of problems in industrial, commercial, medical, networking and educational application domains are being solved by agent-based solutions [2]. The key abstraction in these solutions is the agent. An "agent" is an autonomous, flexible and social system that interacts with its environment in order to satisfy its design agenda. In some cases, two or more agents should interact with each other in a multi agent system (MAS) to solve a problem that they cannot handle alone.

Agent-oriented software engineering (AOSE) is a new discipline that encompasses necessary methods, techniques and tools for developing agent-based systems. It is a powerful way of approaching complex and large scale software engineering problems and developing agent-base systems. Several AOSE methodologies were proposed for developing software, equipped with distinct concepts and modeling tools, in which the key abstraction used in its concepts is that of an agent [3]. Few AOSE methodologies were listed below.

1. MAS CommonKADS (1996-1998)
2. MaSE(1999)
3. GAIA(2000)
4. MESSAGE(2001)
5. TROPOS(2002)
6. PROMETHEUS(2002)
7. ADLEFE(2002)
8. INGENIAS(2002)
9. PASSI(2002)
10. AOR Modeling(2003)

When we analysed and compared the strengths and weaknesses of the above mentioned AOSE methodologies, the strong weakness that we observed from almost all the methodologies were, there is no proper testing mechanism for testing the agent-oriented software [4]. Our works aims to incorporate testing mechanism in Prometheus methodology

II. Prometheus Background

Prometheus is intended to be a practical methodology which is complete and detailed. Prometheus is sufficiently complete such that it covers a range of activities from requirements specification through to detailed design; and it is been sufficiently detailed in that it provides detailed guidance on how to perform the various steps that form the process of Prometheus. The Prometheus methodology includes a description of *concepts* for designing agents, a *process*, a number of *notations* for capturing designs, as well as many "tips" or *techniques* that give advice on how to carry out the steps of Prometheus' process. In Prometheus, an agent's interface with its environment is expressed in terms of *percepts* and *actions.* Proactive agents pursue *goals*, and reactive agents respond to *events* ("significant occurrences"). Agents have *beliefs* and *plans*. Finally, social agents use *messages* to communicate, and these messages are collected in *interaction protocols*.

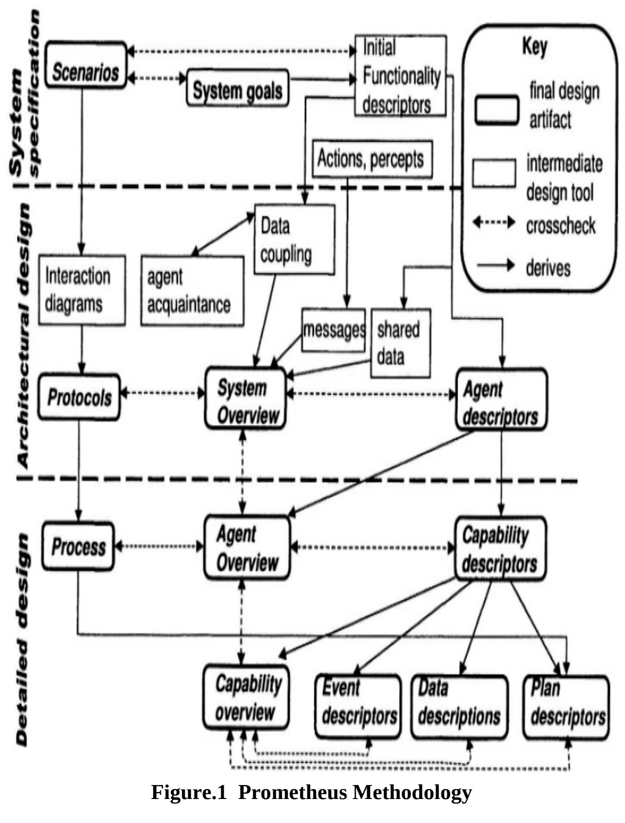

Figure.1 Prometheus Methodology

1. The *system specification phase* focuses on *(i)* identifying the system's *interface,* that, since we are dealing with situated agents, consists of *percepts* (information from the environment), and *actions*; and *(ii)* determining the system's goals, functionalities, and use case scenarios, along with any important shared data. The outputs from this phase are a set of functionality descriptions, percept and action descriptions, system goals, and use case scenarios.
2. The *architectural design phase* uses the outputs from the previous phase to determine which agents the system will contain, how they will interact, and what significant events occur in the environment. The outputs of this phase are a system overview diagram, agent descriptions, agent interaction protocols and a list of significant events and messages between agents.
3. The *detailed design phase* looks at the internals of each agent and how it will accomplish its tasks within the overall system. The outcomes of this phase are detailed diagrams showing the internal functionality of each agent and its capabilities, process diagrams that show the internal processing of the agent, as well as descriptions of data structures used by the agent, plans and subtasks and the details of plan triggers.

Each of these phases includes models that focus on the *dynamics* of the system, (graphical) models that focus on the structure of the system or its components, and textual descriptor forms that provide the details for individual entities.

Testing is an important step in software development in order to assure the correctness of software. Although there are some works on agent oriented testing [1], [6], [7], [8], [10], this activity is often disregarded in most agent oriented methodologies including Prometheus methodology. One reason for this may be that these methodologies mainly focus on analysis and design, as they consider that implementation and testing issues can be performed using well established techniques, mainly from object-oriented software engineering [2]. However, there are relevant features of the agent paradigm that are not yet covered by those more traditional techniques. For instance, autonomy, proactivity, and interactions of agents.

III. EXISTING WORK

Unit Testing recently incorporated into the Prometheus Methodology [5], which tests the plans, events that are handled by multiple plans, and plans that form cyclic dependencies. For instance, plans are triggered by events, an event may be handled by more than one plan, and plans may generate events that trigger other plans either in sequence or in parallel and so on. They presented an overview of the testing process and mechanisms for identifying the order in which the units are to be tested and for generating the input that forms test cases.

They have developed a testing framework [5], which automatically generates and executes unit test cases for an agent system based on its design model (developed in PDT). The testing framework is based on the notion of model based testing [11] which proposes that testing be in some way based on design models of the system. The Prometheus methodology has well developed structured models that are suitable as a basis for model based testing. The design model provides information against which the implemented system can be tested, and also provide an indication of the kind of faults that one might discover as part of a testing process.

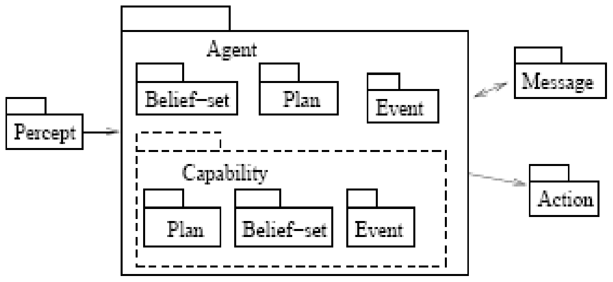

Figure.2 Agent Component Hierarchy in Prometheus

Figure.2 [5] outlines the components of an agent within the Prometheus design. An agent may consist of plans, events and belief-sets, some of which may be encapsulated into capabilities. Percepts and incoming messages are inputs to the agent, while actions and outgoing messages are outputs from the agent. They identify plans, events and belief-sets as the units subject to testing. In their current implementation they do not test belief-sets, which is left for future work.

In many agent systems paradigms (including BDI - Belief, Desire, and Intention [12]) there is a concept of an *event* which triggers selection of one of some number of identified plans, depending on the situation. If one of these plans is actually never used, then this is likely to indicate an error. The concepts of event and plan, and the relationships between them are part of typical agent designs, and can thus be used for model based testing of agent systems. Effective testing of an agent system needs to take account of these kinds of relationships.

IV. PROPOSED WORK

After agent has been unit-tested [3], [5], we have to test its integration with existing agents. In some circumstances, we have to test also the integration of that agent with the agents that will be developed and integrated subsequently. Integration testing involves making sure an agent works properly with the agents that have been integrated before it and with the "future" agents that are in the course of agent testing or that are not ready to be integrated.

In Integration testing we have to test the interaction of agents, communication protocol and semantics, interaction of agents with the environment, integration of agents with shared resources. Observe emergent properties, collective behaviours; make sure that a group of agents and environmental resources work correctly together.

In design of the system we identify a set of collaborative goals. For each of these goals we identify agents that are involved, interaction scenarios, and protocols. Then, we identify fulfillment criteria for the goal. Finally, for each scenario we can define a test suite making use of data identified, i.e. agents, protocols, criteria, and so on.

Integration test suite derivation takes place once we have finished detailed design, so that we can make use of the interaction protocol design. The derivation for collaborative goals consists of the following steps [14]:

```
1. for all g Є {collaborative goals} do
   /* create a test suite for g */
2. identify agents involved
3. identify interaction scenarios
4. identify interaction protocols
5. identify fulfillment criteria (oracle) for each scenario
6. create a test suite for each scenario
7. end for
```

The procedure reads: in the architectural design of the system we identify a set of collaborative goals. For each of these goals we identify agents that are involved, interaction scenarios, and protocols. Then, we identify fulfillment criteria for the goal. Finally, for each scenario we can define a test suite making use of data identified, i.e. agents, protocols, criteria, and so on.

Testing the integration of agents with the operating environment consists of testing their perception and affecting capabilities. That is, we need to make sure that the agents under test are able to perceive changes regarding the resources they are interested in. We test whether they can affect such resources properly. The following steps guide us when deriving test suites for testing the agent-environment interaction:

```
1. for all agent do
2. identify related resources
3. identify integration scenarios
4. identify access policy, interaction protocol, and other related factors if any.
5. identify fulfillment criteria (oracle) for each scenario
6. create one test suite for each scenario
7. end for
```

The procedure is described as follows: for each agent type in the system we identify resources that the agents of the type use. Then, we identify usage or interaction scenarios, access policies, protocols, and other related factors. Finally, we define criteria for each scenario and create a test suite for it, making use of the data identified.

V. Case Study

In the following sections we illustrate the process of design using PDT [4], showing the artifacts produced for the example Small Library Management system [13]. This system should be able to do the following things:

- Allow for validate members, Checkout member details.
- Allow for checkout of books, providing a return date to the customer
- Allow for return of books
- Allow for reservation of unavailable books
- Allow for notification of overdue books
- Allow for notification of arrival of reserved books

An *Equivalence Class* (EC) [5], [9] is a set of input values such that if any value is processed correctly (or incorrectly), then it is assumed that all other values will be processed correctly (or incorrectly). We consider the open intervals and the boundary values of the variable domains to generate ECs, as the former gives equivalent valid values and the latter are *edge* values that should be checked carefully during testing. We also consider some invalid values.

An EC that we define has five fields:

1. *Var-name*: The name of the variable.
2. *Index*: A unique identifier.
3. *Domain*: An open interval or a concrete value.
4. *Validity*: Whether the domain is valid or invalid.
5. *Sample*: A sample value from the domain. Example for open interval (0.0, +∞) and concrete value (x=3). Table 1 gives the equivalence classes for the example variables above.

Electronic Bookstore has been unit tested [5], test case is generated for the *stock manager* agent, in this paper the sample system that we used, was the Simple *Library Management* system. This is an agent-based system *which* contains the agents such as *CheckMember, Checkout, Overdue, and Reservation* agents.

In architectural design, system overview diagram has been shown in figure 3. In unit testing we generate the test case for Checkout agent, in integration testing we test the interaction between two agents such as *Check Member, Checkout.*

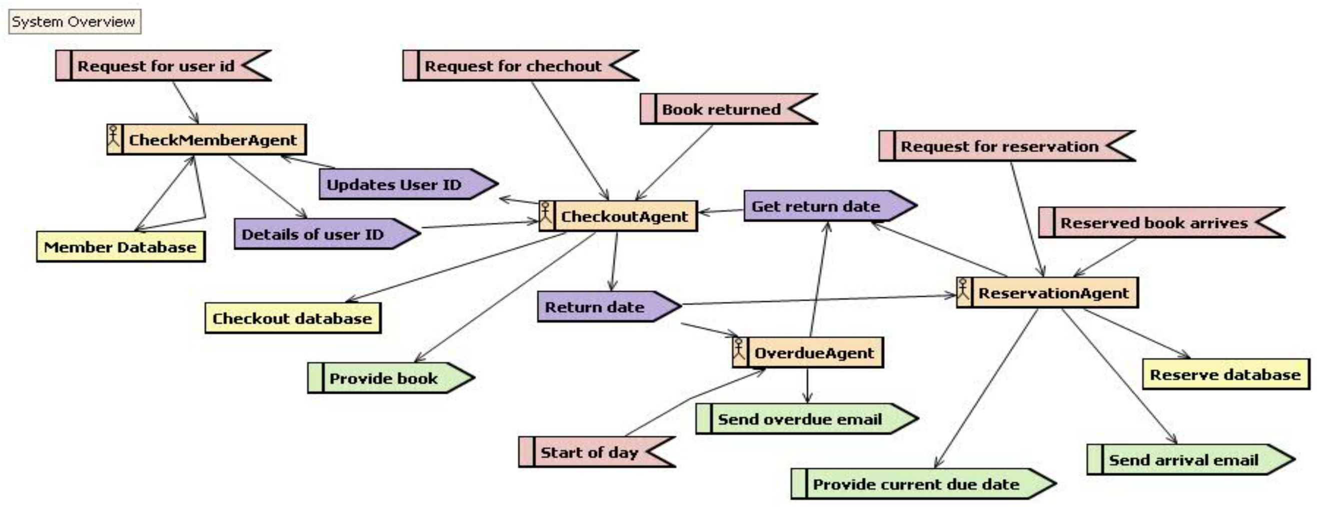

Figure.3 System Overview Diagram

We used the *CheckoutAgent* as the agent under test (AUT), and specifically edited the code to introduce all identified types of faults. The testing framework generator automatically generated the testing framework for the testable units of the *Checkout Agent*, and then executed the testing process for each unit following the sequence determined by the testing-order algorithm [5]. For each unit, the testing framework ran one test suite, which was composed of a set of test cases, with each case having as input one of the value combinations determined.

TABLE I. EC FOR ALL VARIABLES

Variable	Index	Domain	Valid	Sample
BookID	EC-1	(1, 3000)	Yes	100
	EC-2	(-∞, 0)	No	-2
	EC-3	(3001, ∞)	No	3003
Availability	EC-1	Yes	Yes	Yes
	EC-2	No	Yes	No
ReturnDate	EC-1	(DD, MM, YY)	Yes	23-10-10
	EC-2	(DD, MM, YY)	No	10-23-10
	EC-3	Null	Yes	Null

In this Library totally 3000 books are available. Each book has its own ID. User can search the books using the *BookID* with in the range 1 to 3000. If it is a valid *BookID* then user can able to check the availability of book, if book available then *CheckoutAgent* will generate return date for the book. If book not available then user will reserve the book. For which ECs are shown in Table 1.

In Table 1 the Variable *BookID* have the four ECs. EC-1 is the valid input, but EC-2, EC-3 and EC-4 are invalid inputs. In EC1, Domain is the concrete value such as (1, 3000), if any input value between this domain means that should be valid. In EC-2 Domain is open interval such as (-∞, 0) so if any input value negative means that should be invalid. In EC-3 Domain is the open interval such as (3001, ∞) if any input value is greater then 3000 means that should be invalid. In EC1 Domain is the concrete value such as (0), if the input value is zero means the input should be invalid.

In Table 1 the variable Availability have two ECs. Both EC-1 and EC-2 are valid. If the book available means the Domain have the value YES otherwise NO.

In Table 1 the variable *ReturnDate* have the three ECs. If the book available means the agent should provide the return date
for that book. Here test whether the return date format is valid or not. Here valid return date format is (DD, MM, YY), and invalid return date format is (MM, DD, YY).

In Table 2 test the combination of Equivalence Class. In which the index 1 and 2 are valid but index 3, 4 and 5 are invalid.

In Table 2 for Index-1 BookID sample value is 100, it is valid because it comes with in domain range (1-3000). Availability is also valid because the book available. The ReturnDate is valid because of the valid table format (MM, DD, YY).

For Index-3 BookID sample value is valid, Availability is also valid but ReturnDate is not valid because of the invalid table format.

For Index-4 BookID sample value -100 is not valid because of the domain (-∞, 0) is not valid in Table-1.

For Index-5 BookID sample value 0 is not valid because of the domain (0) is not valid in Table-1.

TABLE II. LIST OF EC COMBINATIONS

Index	*BookID*	Availability	ReturnDate	Validity
1	EC-1 (100)	EC-1 Yes)	EC-1 (23-10-10)	Valid
2	EC-1 (100)	EC-2 (No)	EC-3 (Null)	
3	EC-1 (100)	EC-1 (Yes)	EC-2 (10-23-10)	Invalid
4	EC-2 (-2)	EC-2 (No)	EC-3 (Null)	
5	EC-3 (3003)	EC-2 (No)	EC-3 (Null)	

VI. INTEGRATION TESTING

Check Member Agent checks validity of Student and Staff IDs before taking the books. It also maintains the Staff and Student information about return date and number of books taken. Staff can take 10 books and student take 6 books for their IDs. Staff User ID starts from 1 to 100 and Student User ID starts from 1000 to 5000.

Checkout Agent checks the availability of books and provides the return date for the books.

First test the Student or Staff ID before taking the books. Then test Checkout BookID is valid or not. If BookID is valid then check the book is available or not, if books are available then user can take the book otherwise reserve the book. Before user can take the book check the *Student User ID* details, how many books already taken. If student user already taken six books means he can't able to take another book. Once book has been taken CheckoutAgent must provide the return date and send the update message to the *CheckmemberAgent.*

Now see the Table 3 in detail. Each Staff and Student has its own ID number.

Staff User ID valid domain range starts from 1 to 100. Staff User ID has six Equivalence Classes in which EC-1, EC-4 are valid and EC-2, EC-3 is invalid. EC-3 sample value is 101 it is invalid because the domain value (101, ∞) is not valid. EC-4 domain is null because if *Student User ID* processing take place means *Staff User ID* should be Null, if *Staff User ID* processing take place means *Student User ID* Processing should be Null.

```
IF UserID > 0 AND UserID < 101
THEN post Event (Staff User ID)
END IF

IF UserID > 1000 AND UserID < 5001
THEN post Event (Student User ID)
END IF
```

Student User ID has four ECs. Each Student User have there own ID range starts from 1000 to 5000. In which also EC-1, EC-4 are valid and EC-2, EC-3 are in valid similar to *Staff User ID.*

CheckOut Book ID has three ECS. Each and every book has its own ID. Totally 3000 books are present in the library. The Domain valid rage starts from 1 to 3000. In EC-1 *CheckOut Book ID* sample value 100 is valid. EC-2 and EC-3 domains are invalid. In EC-2 domain value starts from (-∞ to 0), the sample input -2 is invalid. In EC-3 sample *CheckOut Book ID* value is 3001, so it is invalid because the *CheckOut Book ID* value must in-between 1 to 3000. For EC-1

```
BookID = 100
IF BookID > 0 AND BookID ≤ 3000
THEN post Event (Check Availability)
```

Availability has two ECs. Here the Domain values are YES and NO. If requested book available means agent would get books, otherwise agent would reserve the book

```
IF Availability = YES
THEN post Event (Request for User ID)
IF Availability = NO
THEN post Event (Reserve Book)
```

Check staff ID before take book has three ECs. Here before providing the books the *CheckOutAgent* request the User Detail message from the *CheckMemberAgent.*

In EC-1 the domain value are (9, 10) where 9 is number of books already taken and 10 is total number of books per user. The sample value denotes number of books requested by the user. In EC-1 the sample input value 1 is valid because the user already taken only 9 books.

```
IF Staff ID books ≤ 10
THEN post Event (Provide Book, Provide Return
    Date, and Update User ID)
```

Check staff ID after take book has four ECs. Here *CheckOutAgent* after providing the books and return date, it will send update message to the *CheckMember Agent.* In EC-1 domain value are (10, 10) where first 10 is the updated value for number of books taken and second 10 is total number of books per user. Here sample values denote total number of books taken including the recently taken books. In EC-2 the sample value is 10, but the domain value (9, 10) is invalid
because the after taking the book the domain value is not updated. In EC-3 domain value are (11, 10) is invalid because user taken 11 books.

```
IF BookID = Provided
THEN Check updated value of user ID
```

Check student ID before take books are similar to the *Check staff ID before take book.*

Check student ID after take book are similar to the Check *staff ID after take books.*

ReturnDate have the three ECs. If the book available means the agent should provide the return date for that book. Here we have to test whether the return date format is valid or not. Here valid return date format is (DD, MM, YY), and invalid return date format is (MM, DD, YY). Now we can see the Table 4 List of Equivalence Class combinations in detail. In which the index 1 and 2 are valid but index 3, 4, 5, 6, 7 and 8 are invalid.

In index-3 *CheckStaff ID after taking the books,* are invalid, because *the* value is not updated.

In index-4 *Student user ID,* sample value 999 and *CheckOut Books* sample value -2 are invalid.

In index-5 Date format is invalid.

In index-6 *CheckStudent ID after taking the books,* are invalid because *books* value is not updated.

In index-7 *Staff user ID,* sample value 101 is not valid,

In index-8 *Student user ID* sample value 5001 is not valid,

VII. CONCLUSION

The need for software testing is well known and accepted. While there are many software testing frameworks for traditional systems like Object-Oriented software systems, there is little work on testing Agent Oriented systems. Most of the work is based on the conformance testing which tests if the system meets the business requirements and restricted to block-box testing. In contrast to these approaches fault-directed testing has been introduced [5] which test the internal processes of the system and not the business requirements, but they concentrated on to the unit- testing, in which they test only internal process of single agents such as events, plans and belief-sets. It is not enough, so we extend this work into integration testing, in which we test the interaction of agents, communication protocol and semantics, interaction of agents with the environment, integration of agents with shared resources. As with the other testing levels, integration test suites are aimed at two distinctive targets:

(i) To refine the interaction design and solve integration problems as early as possible.

(ii) To test the integration of the implemented agents with one another and with the environment.

Roles are the essential concept with agent-oriented software engineering (AOSE). Roles specifications are the first artifacts created by many methodologies. So in future we planned to test the Roles in the Prometheus methodology.

TABLE III. EC FOR ALL VARIABLES

Variable	Index	Domain	Valid	Sample
Staff User ID	EC-1	(1,100)	Yes	99
	EC-2	(-∞, 0)	No	-99
	EC-3	(101, ∞)	No	101
	EC-4	Null	Yes	Null
Student User ID	EC-1	(1000, 5000)	Yes	1001
	EC-2	(-∞, 999)	No	999
	EC-3	(5001, ∞)	No	5001
	EC-4	Null	Yes	Null
CheckOut Book ID	EC-1	(1, 3000)	Yes	100
	EC-2	(-∞, 0)	No	-2
	EC-3	(3001, ∞)	No	3001
Availability	EC-1	Yes	Yes	Yes
	EC-2	No	No	No
Check staff ID before take the book	EC-1	(9, 10)	Yes	1
	EC-2	(10,10)	No	1
	EC-3	Null	Null	Null
Check staff ID after take the book	EC-1	(10, 10)	Yes	10
	EC-2	(9, 10)	No	10
	EC-3	(11, 10)	No	11
	EC-4	Null	Null	Null
Check student ID before take the book	EC-1	(5, 6)	Yes	1
	EC-2	(6, 6)	No	1
	EC-3	Null	Null	Null
Check student ID after take the book	EC-1	(6, 6)	Yes	6
	EC-2	(5, 6)	No	6
	EC-3	(7, 6)	No	7
	EC-4	Null	Null	Null
Provide Return Date	EC-1	(DD, MM,	Yes	23-10-10
	EC-2	(DD, MM,	No	10-23-10
	EC-3	Null	Null	Null

TABLE IV. LIST OF EC COMBINATIONS

Index	**Staff UserID**	**Student UserID**	**Check Out Book**	**Availability**	**Check Staff ID before taking the book**	**Check Staff ID after taking the book**	**Check Student ID Before taking the book**	**Check Student ID after taking the book**	**Provide Return Date**	**Validity**
1	EC-1 (99)	Null	EC-1 (100)	EC-1 (Yes)	EC-1 (1)	EC-1 (10)	Null	Null	EC-1 (23-10-10)	Valid
2	Null	EC-1 (1001)	EC-1 (100)	EC-1 (Yes)	Null	Null	EC-1 (1)	EC-1 (6)	EC-1 (23-10-10)	
3	EC-1 (99)	Null	EC-1 (100)	EC-2 (Yes)	EC-1 (1)	EC-1 (9)	Null	Null	EC-1 (23-10-10)	Invalid
4	Null	EC-2 (999)	EC-2 (2)	EC-2 (No)	Null	Null	EC-1 (1)	EC-1 (5)	EC-1 (23-10-10)	
5	EC-1 (99)	Null	EC-1 (100)	EC-1 (Yes)	EC-1 (1)	EC-1 (9)	Null	Null	EC-2 (10-23-10)	
6	Null	EC-1 (1001)	EC-1 (100)	EC-1 (Yes)	Null	Null	EC-1 (1)	EC-1 (4)	EC-2 (23-10-10)	
7	EC-3 (101)	Null	EC-1 (100)	EC-1 (Yes)	EC-1 (1)	EC-1 (9)	Null	Null	EC-1 (23-10-10)	
8	Null	EC-3 (5001)	EC-1 (100)	EC-1 (Yes)	Null	Null	EC-1 (1)	EC-1 6)	EC-1 (23-10-10)	

REFERENCES

[1] Mailyn Moreno, Juan Pavón, Alejandro Rosete. "Testing in Agent Oriented Methodologies", S.Omatu et al. (Eds.): IWANN 2009, Part II, LNCS 5518, pp. 138–145, 2009. Springer-Verlag Berlin Heidelberg 2009.

[2] Praveen Ranjan Srivastava, Karthik Anand V, Mayuri Rastogi, Vikrant Yadav, G Raghurama. "Extension of Object-Oriented Software Testing Techniques to Agent Oriented Software Testing", in *Journal of Object Technology*, vol. 7, no. 8, pp. 155-163, November-December 2008.

[3] Zhiyong Zhang, John Thangarajah and Lin Padgham, "Automated Unit Testing Intelligent Agents in PDT (Demo Paper)". Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril,Portugal,pp. 1673-1674, May,12-16.,2008.

[4] .Padgham, J.Thangarajah, and M.Winikoff. "The Prometheus Design Tool – A Conference Management System Case Study". AOSE 2007, LNCS 4951, pp.197–211, 2008. Springer-Verlag Berlin Heidelberg 2008.

[5] Z. Zhang, J.Thangarajah, and L.Padgham. "Automated unit testing for agent systems". In 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 07), pages 10–18, Spain, July 2007.

[6] Tiryaki, A.M., Öztuna, S., Dikenelli, O., Erdur, R.C.: SUNIT. "A Unit Testing Framework for Test Driven Development of Multi-Agent Systems". In: Padgham, L., Zambonelli, F. (eds.) AOSE VII / AOSE 2006. LNCS, vol. 4405, pp. 156–173. Springer, Heidelberg (2007).

[7] Coelho, R., Kulesza, U., Staa, A.v., Lucena, C. "Unit Testing in Multi-Agent Systems Using Mock Agents and Aspects". In: International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, pp. 83–90. ACM, Shanghai (2006).

[8] Rouff, C. "A Test Agent for Testing Agents and Their Communities". In: Aerospace Conference Proceedings, vol. 5, pp. 2633–2638 (2002).

[9] Binder, R. V. (1999). "*Testing Object-Oriented Systems: Models, Patterns, and Tools*". Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[10] Nwana, H., Ndumu, D., Lee, L., Collis, J.ZEUS. "A Toolkit for Building Distributed Multi-Agent Systems". Applied Artificial Intelligence 13, 129–185 (1999).

[11] L. Apfelbaum and J. Doyle. "Model Based Testing". In the 10th International Software Quality Week Conference, CA, USA, 1997.

[12] Rao, A. S. and Georgeff, M. P. "BDI Agents: From Theory to Practice". In Lesser, V., editor, *the First International Conference on Multi-Agent Systems*, pages 312–319, San Francisco (1995).

[13] http://www.cs.rmit.edu.au/agents/pdt/docs/Tutorial.pdf

[14] Cu Duy Nguyen. "Testing Techniques for Software Agents". PhD Dissertation, International Doctorate School in Information &Communication Technologies, DIT - University of Trento, December 2008.